山东大学
SHANDONG UNIVERSITY

崇新学堂

2024 — 2025 学年第一学期

实验报告

课程名称：Introduction to EECS Lab

实验名称：DL6 - Sizable Following

学生姓名：胡君安、陈焕斌、黄颢

实验时间：2024 年 11 月 7 日

Design Lab 6 - Sizable Following

1. Introduction

Last week, we used a proportional controller to move a robot parallel to a wall, aiming to maintain a constant desired distance. The forward velocity was set to 0.1 m/s, and the angular velocity was proportional to the error signal, the difference between the desired and current distances. However, no proportionality constant provided good performance. Large constants caused fast oscillations, while small constants resulted in large errors, especially when the robot's initial angle was not parallel to the wall. This week, we will develop two new controllers: one considering both the previous and current distances to the wall, and the other considering the current angle and distance to the wall.

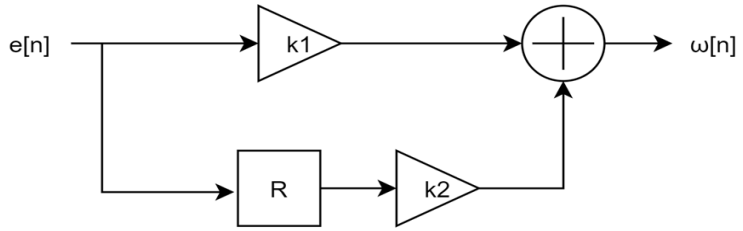
2. Experimental step

Step 1: Writing the delayPlusPropModel Code

Check Yourself 1

Draw a block diagram of the controller.

The value of the input controller needs to be multiplied by k1 first, delayed, multiplied by k2, and then added together.The block diagram of the controller is as follows.



Referring to the above diagram, we have written the following codes.

```
Python
def delayPlusPropModel(k1, k2):
    T = 0.1
    V = 0.1
    controller = sf.FeedforwardAdd(sf.Gain(k1), sf.Cascade(sf.Gain(k2),
sf.R()))
    plant1 = sf.Cascade(sf.Cascade(sf.Gain(T), sf.R()),
sf.FeedbackAdd(sf.Gain(1), sf.R()))
    plant2 = sf.Cascade(sf.Cascade(sf.Gain(T * V), sf.R()),
sf.FeedbackAdd(sf.Gain(1), sf.R()))
```

```
    # Combine the three parts
    sys = sf.FeedbackSubtract(sf.Cascade(sf.Cascade(controller, plant1),
plant2))
    return sys
```

Check Yourself 2

Here are graphs of two functions. Use optimize.optOverLine to find the minimum of one of them. The function f has a minimum at x = 0.5 of value −0.25; the function h has a minimum at x = 1.66 with value −0.88.

The relevant code is as follows.

```
Python
optimize.optOverLine(lambda x: x**2 - x, -1, 1, 100)
```

```
>>> optimize.optOverLine(lambda x: x**2 - x, -1, 1, 100)
(-0.25, 0.50000000000000078)
```

```
Python
optimize.optOverLine(lambda x: x**5 - 7*x**3 + 6*x**2 + 2, -1, 10, 100)
```

```
>>> optimize.optOverLine(lambda x: x**5 - 7*x**3 + 6*x**2 + 2, -1, 10, 100)
(-0.87533301759999915, 1.6400000000000003)
```

Step 2: Finding Appropriate Values

Based on the example of solving the maximum and minimum values above, we can write the following code.

```
Python
def bestk2(k1, k2Min, k2Max, numSteps):
    print(optimize.optOverLine(lambda k2: abs(delayPlusPropModel(k1,
k2).dominantPole())
                                       , k2Min, k2Max, numSteps))
```

Through the code, we solved k2 and the magnitude of dominant pole.

```
bestk2
(0.99498743841471793, -9.9499999999999993)
(0.98466966344277818, -29.749999999999996)
(0.94556460913992268, -97.350000000000151)
(0.77229854893668504, -271.69999999999357)
```

The result of bestk2

The relevant data is as follows.

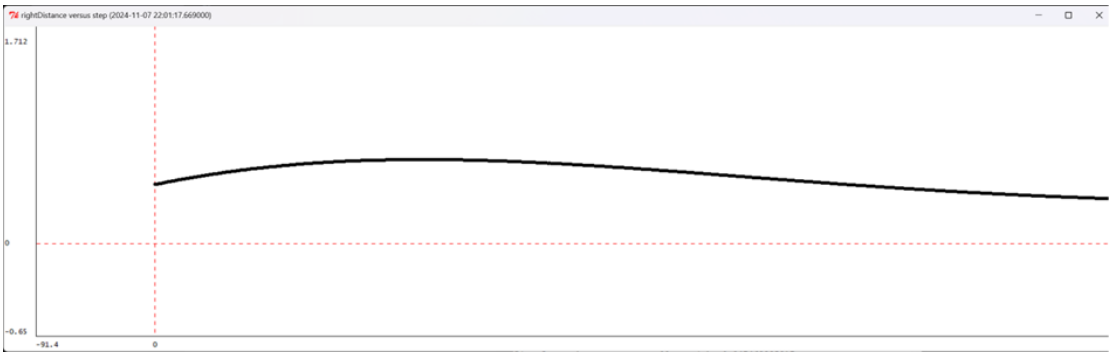
k1	k2	magnitude of dominant pole
10	-9.95	0.995
30	-29.75	0.985
100	-97.35	0.946
300	-271.70	0.772

Step 3: Writing the WallFollower Code

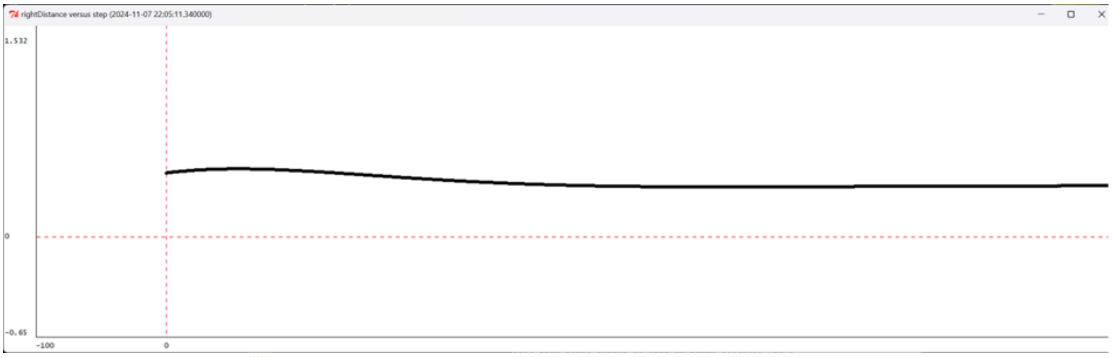
The code for the WallFollower section is as follows.

```
Python
class WallFollower(sm.SM):
    startState = desiredRight
    def getNextValues(self, state, inp):
        e1 = desiredRight - inp
        e2 = desiredRight - state
        w = k1*e1 +k2*e2
        return (inp,io.Action(fvel = forwardVelocity,rvel = w))
```

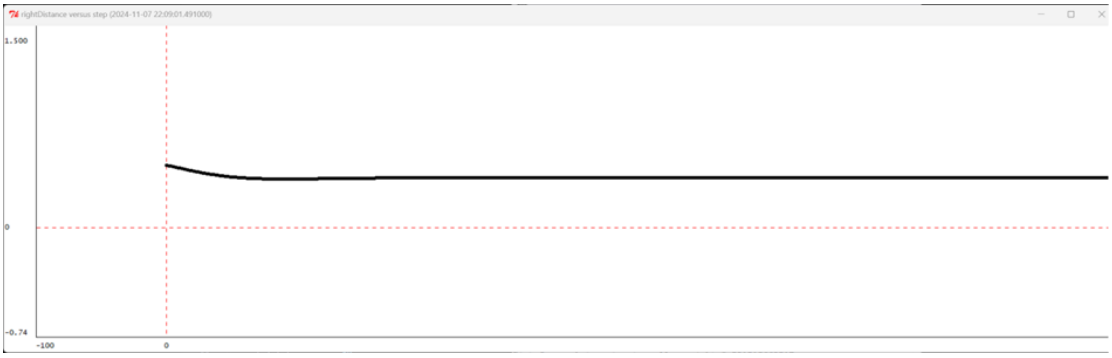
Step 4: Analogue Simulation



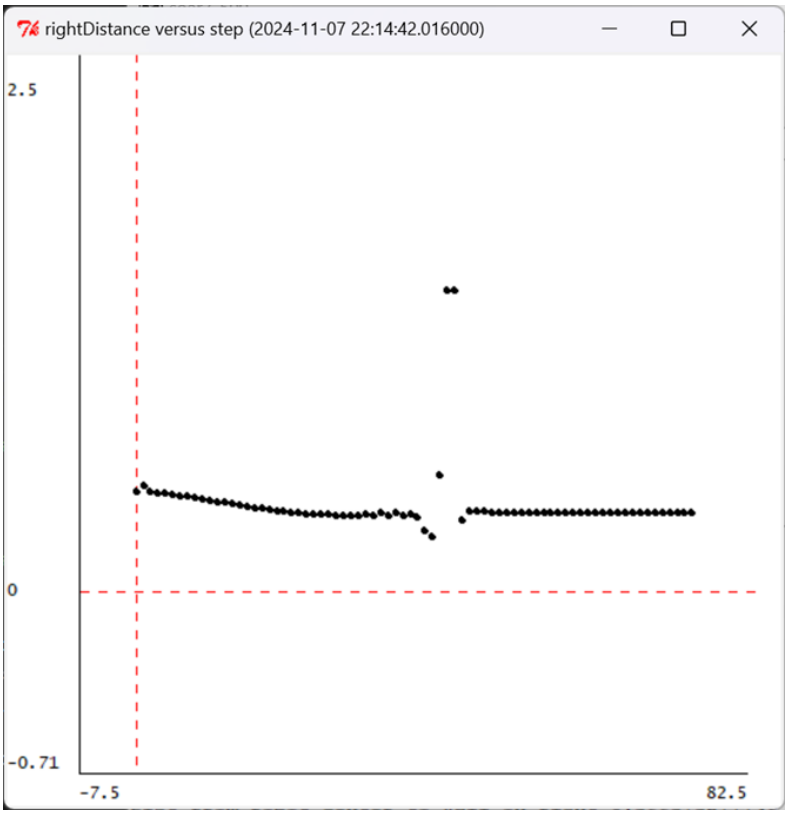
k1 =10, k2 = -9.95



k1 = 30, k2 = -29.75



k1 = 100, k2 = -97.35



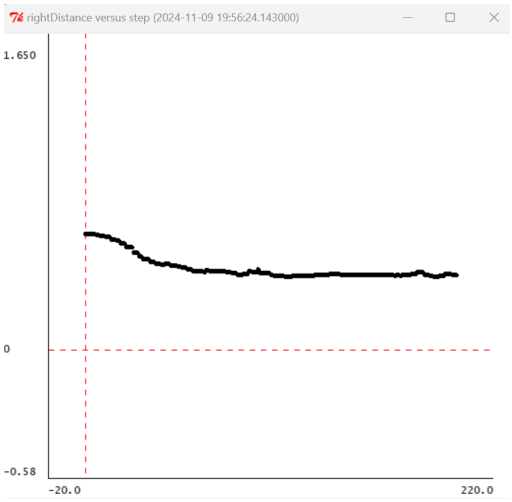
k1 = 300, k2 = -271.70

Check Yourself 3

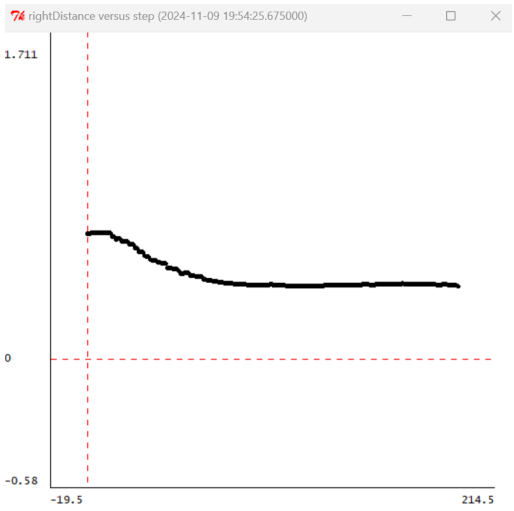
Which of the four gain pairs work best in simulation? Which gains cause bad behavior?

The best results are achieved when k1=100 and k2=-97.35, while bad behavior occurs when k1=300 and k2=-271.70.

Step 5: Actual Test 1



k1 = 300, k2 = -271.70



k1 = 100, k2 = -97.3

Check Yourself 4

Which of the four gain pairs work best on a robot?

The best results are achieved when k1 = 100, k2 = -97.35.

Are the best gains the same as in simulation?

Yes, they are.

Which gains cause bad behavior?

In actual control, the k will cause too much vibration when it's too high, which may lead to hitting the wall. The picture above shows the vibration.

Checkoff 1.

Show a staff member plots for the simulated and real robot runs, and discuss their relationship.

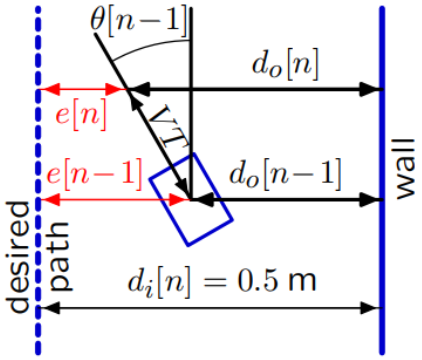
The difference between them may occur when the k is high. In simulation, after an intense vibration, the curve returns to stability. Whereas on the robot, the curve continues to vibrate.

How is the robot's behavior related to the magnitude of the dominant pole, for each of the gain pairs? Explain how you chose the starting state of your controller.

Although the larger the absolute value of the gain pair is, the faster convergence speed is, it is easier for the car to oscillate in the experiment.

Step 6: Building the Angle Plus Propotional Model

In this problem, the difference (e[n] - e[n-1]) can be interpreted in terms of the angle (θ[n-1]). Therefore, a delay-plus-proportional controller can base the next angular velocity on both the robot's position and angle. Note that the position information is for time (n), while the angle information is for time (n-1).



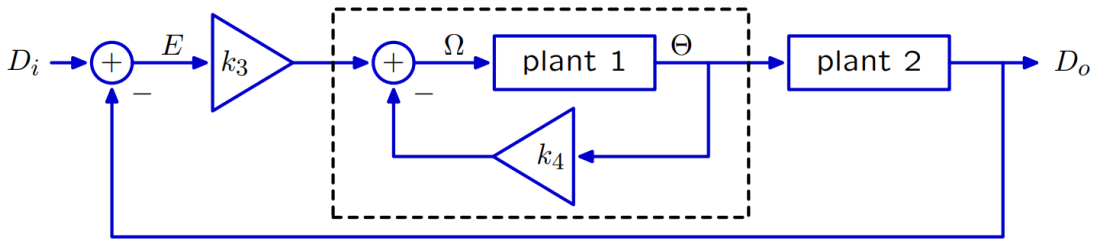
We can use the following equation to describe the system:

$$\omega[n] = k_3(d_i[n] - d_o[n]) + k_4(\theta_d - \theta[n])$$

where θd = 0 represents the desired angle (as measured relative to the wall), so that

$$\omega[n] = k_3 e[n] - k_4 \theta[n]$$

The whole system diagram is:



In designLab06Work.py, create a Python procedure anglePlusPropModel that takes gains k3 and k4 as inputs and returns a SystemFunction for angle-plus-proportional control. Assume (T = 0.1) seconds and (V = 0.1) m/s.

```
Python
def anglePlusPropModel(k3, k4):
    T = 0.1
    V = 0.1

    plant1 = sf.Cascade(sf.Cascade(sf.Gain(T), sf.R()),
sf.FeedbackAdd(sf.Gain(1), sf.R()))
    plant2 = sf.Cascade(sf.Cascade(sf.Gain(T * V), sf.R()),
sf.FeedbackAdd(sf.Gain(1), sf.R()))
    # The complete system
    sys =
sf.FeedbackSubtract(sf.Cascade(sf.Cascade(sf.Gain(k3),sf.FeedbackSubtract(
plant1,sf.Gain(k4))),plant2))

    return sys
```

For k3 equal to 1, 3, 10, and 30, determine values of k4 that minimize the magnitude of the least stable pole of the angle-plus-proportional system.

```
Python
def bestk4(k3, k4Min, k4Max, numSteps):
    print (optimize.optOverLine(lambda k4: abs(anglePlusPropModel(k3,
k4).dominantPole())
                                          ,k4Min, k4Max, numSteps))

print ('bestk4')
bestk4(1, -10, 10, 400)
bestk4(3, -10, 10, 400)
bestk4(10, -10, 10, 400)
bestk4(30, -30, 30, 1200)
```

k3	k4	magnitude of dominant pole
1	0.60	0.97
3	1.05	0.95

10	2.00	0.90
30	3.45	0.83

Step 7: Finishing the Angle Plus Propotional Brain

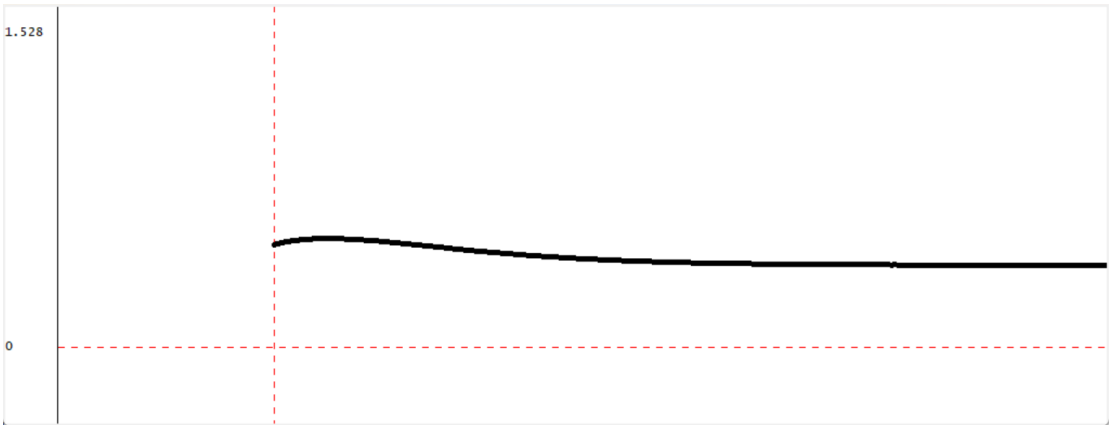
Edit the WallFollower state machine class in anglePlusPropBrainSkeleton.py to implement the proportional plus angle controller. The brain consists of two parts in cascade:

1. **Sensor Class**: A state machine with SensorInputs as input, outputting pairs of the perpendicular distance to the wall on the right and the angle to the wall.

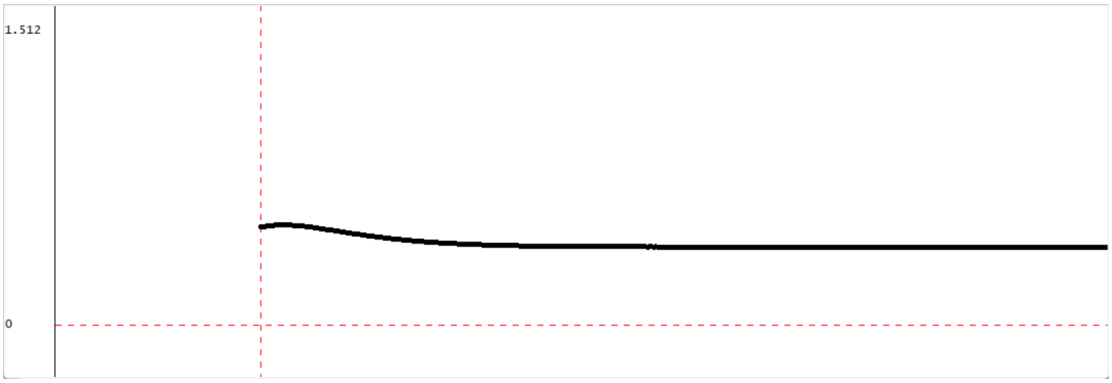
2. **WallFollower Class**: A state machine with the distance and angle pair as input, outputting an instance of io.Action.

If sonar 6 or 7 is out of range, the angle cannot be calculated, and the Sensor machine will output None for the angle. In such cases, set the angular velocity to 0.

```
Python
desiredRight = 0.4
forwardVelocity = 0.1

# No additional delay.
# Output is a sequence of (distance, angle) pairs
class Sensor(sm.SM):
   def getNextValues(self, state, inp):
       v = sonarDist.getDistanceRightAndAngle(inp.sonars)
       print 'Dist from robot center to wall on right', v[0]
       if not v[1]:
           print '******  Angle reading not valid  ******'
       return (state, v)

# inp is a tuple (distanceRight, angle)
class WallFollower(sm.SM):
    startState = None
    def getNextValues(self, state, inp):
        (currentDist, angle) = inp
        if inp[1]==None:
            w = 0
        else:
            e1 = desiredRight - currentDist
            e2 = -angle
            w = k3*e1 + k4*e2
        return (state ,io.Action(fvel = forwardVelocity,rvel = w))

k3 = 10
k4 = 2
sensorMachine = Sensor()
sensorMachine.name = 'sensor'
mySM = sm.Cascade(sensorMachine, WallFollower())
```



k3=1, k4=0.6



k3=3, k4=1.05

k3=10, k4=2

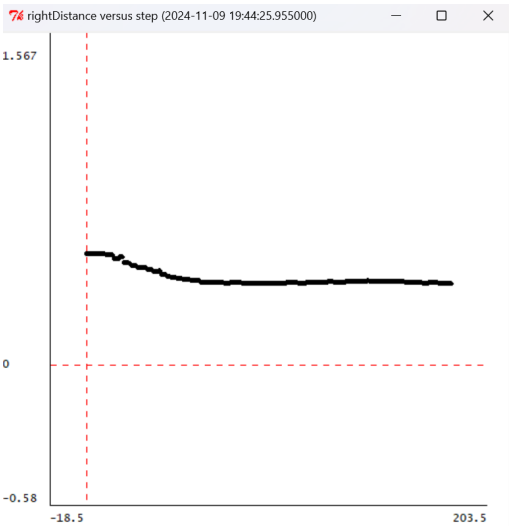
k3=30, k4=3.45

Check Yourself 5

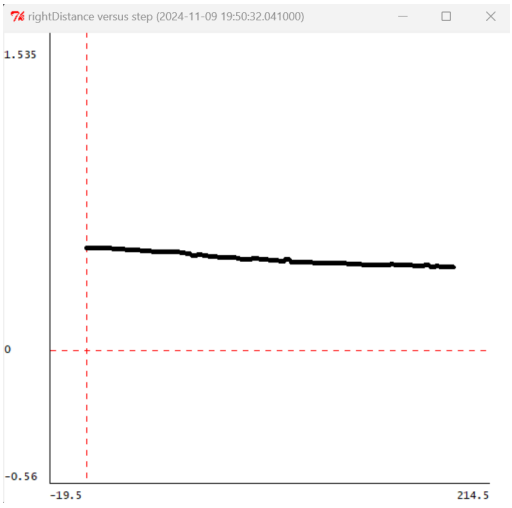
Which of the four gain pairs work best in simulation?

The best results are achieved when k3=30 and k4=3.45.

Step 8: Actual Test 2



k3=30, k4=3.45



k3=1, k4=0.6

Check Yourself 6

Which of the four gain pairs work best on a robot?

The best results are achieved when k3=10 and k4=2.

Are the best gains the same as in simulation?

No.

Which gains cause bad behavior?

In actual control, the k will cause vibration when it's too high

Checkoff 2.

Show a staff member plots for the simulated and real robot runs, and discuss their relationship.

In simulation, with time pass by, the curve becomes straight. Whereas on the robot, the curve remains some small vibrations.

How is the robot's behavior related to the magnitude of the dominant pole, for each of the gain pairs? Explain how you chose the starting state of your controller.

Although the larger the absolute value of the gain pair is, the faster convergence speed is, it is easier for the car to oscillate in the experiment.

Which controller(delay-plus-proportional or angle-plus-proportional) performs better? Explain why?

According to the curves above, we determined that the angle-plus-proportional is better. Angle-plus-proportional reduces the order of the denominator of the system function and reduces the delay, which leads to better performance than delay-plus-proportional.

3. Summary

1. In this experiment, we enhanced the system used in the previous two experiments, enabling the car to move along the wall more smoothly.

2. During the first simulation, although the best simulation outcome emerged, the actual performance was the worst. After discussion, we attributed this to the inability of the real-world physical environment to synchronize the transmission of current and delay values.

3. For the real car, it's necessary to take into account that the real - world physical system differs from the simulation. Theoretically, a smaller pole leads to easier convergence. However, there is a certain delay in the current system, which can cause oscillations. We achieved control of fast convergence and avoidance of severe oscillations through proportional gain and feed forward.